

PROGRAMMING EXPERT

FAQ

Q What is the open source movement?

A It's a global community of programmers who believe in free software. The term 'open source' means that the source code of a project is publicly available. It first appeared when Netscape released the source code of its Navigator web browser, although in effect Linus Torvalds' earlier Linux project was open source too. Far from being an amateurish free for all, open source development is tightly coordinated, and each volunteer works on a small part of the code. The whole project is built regularly to create releases that are tested and improved. Stable versions are distributed as free software.

SourceForge (<http://sourceforge.net>) is home to many open-source projects, but the most important ones, such as Linux, Mozilla, Samba and Mono, have their own websites.

Mike James

IT author, lecturer
and programmer

mike@computershopper.co.uk

You have mail

Want to be alerted to that all-important email? Mike James explains how to make your own You Have Mail application



If you believe what you see in films or on TV, the arrival of an email is always accompanied by some extravagant display; at the very least, the words You Have Mail (YHM) loom large onscreen. This month's project started out with the aim of making the YHM movie experience a reality, but then it branched out into something more useful.

If you have ever waited for an important email, you may have experienced the frustration of having to check your mailbox every few minutes. The YHM application is designed to do this for you. It isn't 100 per cent complete; you can decide for yourself how to specify the email you are waiting for, and exactly what happens when it arrives. However, the basic mechanism for scanning an Outlook Express mailbox is fully described. Just add your own code to, say, send an SMS, ring a bell or display an over-the-top multimedia animation to signal the email's arrival.

The project works with Outlook Express – not full Outlook – and has been tested on Windows XP, but it should work with Vista's Windows Mail, the new name for Outlook Express. It doesn't work under Windows XP 64-bit, though, because of problems with COM Interop.

BRIGHTER OUTLOOK

This project uses the Outlook Express API or OEAPI. This is an old-fashioned COM interface-based API that doesn't provide any ease-of-use features. You'll find its documentation by searching the Microsoft website for Windows Mail API or for IStoreNamespace. In *Shopper* August 2006, we looked at how to convert this API into a C# object that we could use from any .NET language. That project simply converted the IStoreNamespace interface into an object because it only needed to inspect the message store and discover what sorts of messages were stored in each folder. Our new project needs to use IStoreNamespace and also IStoreFolder.

We won't go over the details of converting an Interface to a class because it's much the same ground we covered in the August issue. However some differences are worth explaining. The biggest is that the IStoreFolder interface is created by one of the functions in the IStoreNamespace interface; this complication is

surprisingly easy to deal with, although some standard techniques proved far from straightforward.

The additional interface, IStoreFolder, is created in the same class file used to create IStoreNamespace, and it is assumed that you have the file that contains the OE class from the project in the August 2006 issue; it's on this month's cover disc along with the rest of the program if you don't. Most of the detailed information about the API and the way everything works can be discovered only by reading the header file msoeapi.h, which is installed in the Includes directory of the Windows SDK. As this is a header file intended for C or C++ programmers, it isn't of direct use. However, by reading it you can find out how the interfaces have been defined. Unfortunately, to interpret a header file you have to know C/C++, but it is similar to C# and you can look up anything you are unsure of.

The first step is to enter the start of the IStoreFolder Interface definition. It doesn't have to be complete at this stage. Just enter the functions that the Interface defines listed in the order they occur:

```
[Guid("E70C92AC-4BFD-11d1-8A95-00C04FB951F3"),
InterfaceType(ComInterfaceType.Interface
IsIUnknown)]
public interface IStoreFolder
{
    void GetFolderProps();
    void GetMessageProps();
    void FreeMessageProps();
    void DeleteMessages();
    void SetLanguage();
    void MarkMessagesAsRead();
    void SetFlags();
    void OpenMessage();
    void SaveMessage();
    void BatchLock();
    void BatchFlush();
    void BatchUnlock();
    void CreateStream();
    void CommitStream();
    void RegisterNotification();
    void UnregisterNotification();
}
```



```
void Compact();
void GetFirstMessage();
void GetNextMessage();
void GetMessageClose();
```

You can't use any of the functions of the interface at this stage, because you have to add the details of their parameters and how these are to be converted, or 'marshalled' in C# jargon, to and from .NET data types. However, you don't have to complete the definitions of all the functions in the interface – just those you want to use. In this case, we need just five. The first is:

```
void GetFolderProps();
```

This gets the details of the folder, such as how many emails it contains, for example. The next two do the same job:

```
void GetFirstMessage();
void GetNextMessage();
```

These get all the details of the relevant message, including who sent it and when. Finally, we need two cleanup utilities, which have to be called to complete any inspection of the messages in a folder:

```
void FreeMessageProps();
void GetMessageClose();
```

Converting the interface specifications to C# is straightforward:

```
void GetFolderProps(
    [In, MarshalAs(UnmanagedType.U4)]
    Int32 dwReserved,
    [Out, MarshalAs(UnmanagedType.Struct)]
    out _FOLDERPROPS pProps);

[PreserveSig]
Int32 GetFirstMessage(
    [In, MarshalAs(UnmanagedType.U4)]
    Int32 dwFlags,
    [In, MarshalAs(UnmanagedType.U4)]
    Int32 dwMsgFlags,
    [In, MarshalAs(UnmanagedType.U4)]
    UInt32 dwMsgIdFirst,
    [In, Out, MarshalAs(UnmanagedType.Struct)]
    ref _MESSAGEPROPS pProps,
    [Out, MarshalAs(UnmanagedType.U4)]
    out UInt32 phEnum);

[PreserveSig]
Int32 GetNextMessage(
    [In, MarshalAs(UnmanagedType.U4)]
    UInt32 hEnum,
    [In, MarshalAs(UnmanagedType.U4)]
    Int32 dwFlags,
    [In, Out, MarshalAs(UnmanagedType.Struct)]
    ref _MESSAGEPROPS pProps);

void FreeMessageProps(
    [In, Out, MarshalAs(UnmanagedType.Struct)]
    ref _MESSAGEPROPS pProps);

void GetMessageClose(
    [In, MarshalAs(UnmanagedType.U4)]
    UInt32 hEnum);
```

The _FOLDERPROPS struct was defined as part of the previous interface, but we need to define a _MESSAGEPROPS structure. This is a straight translation of the C/C++ structure:

```
[StructLayout(LayoutKind.Sequential,
    Pack = 1, CharSet = CharSet.Ansi)]
public struct _MESSAGEPROPS
{
    public Int32 cbSize;
    public Int32 dwReserved;
    public Int32 dwMessageId;
    public Int32 dwLanguage;
    public Int32 dwState;
    public Int32 cbMessage;
    public Int32 priority;
    public System.Runtime.InteropServices.ComTypes.FILETIME ftReceived;
    public System.Runtime.InteropServices.ComTypes.FILETIME ftSent;
    [MarshalAs(UnmanagedType.LPStr)]
    public string pszSubject;
    [MarshalAs(UnmanagedType.LPStr)]
    public string pszDisplayTo;
    [MarshalAs(UnmanagedType.LPStr)]
    public string pszDisplayFrom;
    [MarshalAs(UnmanagedType.LPStr)]
    public string pszNormalSubject;
    public Int32 dwFlags;
    public Int32 pStmOffsetTable;
};
```

WRAPPING THE INTERFACE

You could just use the Interface defined above as it stands, but it is sensible to make it easier to use by 'wrapping' it in a new class: OEFolder. The complication is that our original OE class, created to wrap the IStoreNamespace interface, contains a method OpenSpecialFolder that creates an OEFolder object. This makes it impossible to create an OEFolder constructor that creates the IStoreFolder interface that the class wraps. The solution is for the original OpenSpecialFolder method to use a two-stage process. First, it creates an instance of the IStoreFolder interface and then uses this in the OEFolder constructor to create an OEFolder object that 'wraps' the interface:

```
public OEFolder OpenSpecialFolder(
    SPECIALFOLDER FolderID)
{
    IStoreFolder ppFolder;
    OEFolders.OpenSpecialFolder(
        (int)FolderID, 0, out ppFolder);
    OEFolder fold = new
        OEFolder(ppFolder);
    return fold;
}

00
```

You should add some error-handling code to this, because sometimes Outlook Express is temporarily unable to open a folder and this will generate an exception.

The constructor is simply:

```
public class OEFolder
{
    private IStoreFolder m_IFolder;

    public OEFolder(IStoreFolder
        folder)
    {
        m_IFolder = folder;
    }
}
```

```
}
```

The member variable m_IFolder points to the IStoreFolder interface passed to the constructor and can be used by the rest of the class to access it.

Our new OEFolder object can represent any of the special Outlook Express folders. The next task is to add methods that wrap the five functions of the IStoreFolder Interface we will use. The first is easy as it is a repeat of the corresponding method in IStoreNamespace:

```
public FOLDERPROPS GetFolderProps()
{
    _FOLDERPROPS FP = new _FOLDERPROPS();
    FP.cbSize = Marshal.SizeOf(FP);
    m_IFolder.GetFolderProps(0, out FP);
    FOLDERPROPS fp = OE.transfer(FP);
    return fp;
}
```

The FOLDERPROPS structure is a more civilised, managed version of the structure returned by the Interface.

The next method we need reads the properties of each message in the folder in turn. The raw Interface does this using two functions. The first, GetFirstMessage, returns the details of the first message and a handle to an enumerator. Subsequent messages are returned by GetNextMessage as long you pass it the handle to the enumerator. There is no need for us to separate getting the first and subsequent messages because we can construct a method that knows if it is to get the first or next according to the value of the enumerator. The method begins:

```
public MESSAGEPROPS GetNextMessage(
    ref UInt32 ehandle)
{
    _MESSAGEPROPS MP = new _MESSAGEPROPS();
    MP.cbSize = Marshal.SizeOf(MP);
    Int32 result=0;
}
```

If the enumerator is zero, we call the function to get the first message:

```
if (ehandle == 0)
{
    result = m_IFolder.
        GetFirstMessage(
            0, 0, MESSAGEID_FIRST,
            ref MP, out ehandle);
}
```

If the enumerator has a non-zero value, we call the function to get the next message:

```
else
{
    result = m_IFolder.
        GetNextMessage(
            ehandle, 0, ref MP);
}
```

We also need a constant:

```
private const UInt32
    MESSAGEID_FIRST = 0xffffffff;
```

After calling the relevant function, the message properties is stored in the MP structure, but it's not in a very C# programmer-friendly form. The solution is to transfer it to a managed struct

APRESS

The Game Maker's Apprentice



£27.99

Jacob Habgood and Mark Overmars have produced a very enjoyable book. Games are a fun way to get started with programming, but it is difficult to create anything that looks impressive. One solution is to use a dedicated game-design program, but then you risk learning about the program and nothing else.

This book is based on Game Maker and a CD copy is bound into the back. It begins with detailed instructions on how to install Game

Maker and get started. From there, it shows you how to create games using drag-and-drop techniques, the ideal way to get quick and exciting results. Unusually, it moves on to more programming-based methods using Game Maker Language (GML) and covers wider programming issues such as AI and logic. It even strays away from action games and into the quieter but instructive coding of noughts and crosses.

This full-colour book looks good, is easy to follow and lots of fun. It should help you acquire many of the basic ideas of programming. From here, you could move on to a language such as C# for mainstream programming, or hone your games programming further by finding out more about game engines.



SUMMARY

- Great fun; easy to use
- Doesn't go far enough

SUPPLIER www.amazon.co.uk
ISBN 1590596153
PAGES 300

MESSAGEPROPS complete with an enumeration for the message state:

```
public struct MESSAGEPROPS
{
    public Int32 MessageId;
    public Int32 Language;
    public MsgState State;
    public Int32 MessageSize;
    public Int32 priority;
    public DateTime Received;
    public DateTime Sent;
    public string Subject;
    public string DisplayTo;
    public string DisplayFrom;
    public string NormalSubject;
    public Int32 Flags;
};

public enum MsgState
{
    MSG_DELETED = 0x0001,
    MSG_UNREAD = 0x0002,
    MSG_SUBMITTED = 0x0004,
    MSG_UNSENT = 0x0008,
    MSG_RECEIVED = 0x0010,
    MSG_NEWMSG = 0x0020,
    MSG_NOSECUI = 0x0040,
    MSG_VOICEMAIL = 0x0080,
    MSG_REPLIED = 0x0100,
    MSG_FORWARDED = 0x0200,
    MSG_RCPTSENT = 0x0400,
    MSG_FLAGGED = 0x0800
}
```

Some of the fields need no processing or are converted using a simple cast:

```
MESSAGEPROPS mp = new
    MESSAGEPROPS();
if (result == 0)
{
    mp.MessageId = MP.dwMessageId;
    mp.Language = MP.dwLanguage;
    mp.State = (MsgState)MP.dwState;
    mp.MessageSize = MP.cbMessage;
    mp.priority = MP.priority;
}
```

The first fields that need some processing are the date and times, which are represented as FileTime structures. These are best converted to DateTime objects, using a function we'll write

later to first convert them to long integers:

```
mp.Received = System.DateTime.
    FromFileTime(
        FileTimeToLong(MP.ftReceived)
    );
mp.Sent = System.DateTime.
    FromFileTime(
        FileTimeToLong(MP.ftSent));
mp.Subject = MP.pszSubject;
mp.DisplayTo = MP.pszDisplayTo;
mp.DisplayFrom = MP.
    pszDisplayFrom;
mp.NormalSubject = MP.
    pszNormalSubject;
mp.Flags = MP.dwFlags;
```

Finally, if the result was non-zero there are no more messages and so we can close the enumeration:

```
else
{
    m_IFolder.GetMessageClose
        (ehandle);
}
```

Finally we free the MessageProps structure:

```
m_IFolder.FreeMessageProps
    (ref MP);
return mp;
```

Now we can write the function to convert FileTime structures to 64-bit integers:

```
private long FileTimeToLong(
    System.Runtime.InteropServices.
        ComTypes.FILETIME ftime)
{
    long h = ((long)ftime.
        dwHighDateTime) << 32;
    long l = 0xFFFFFFFF & ftime.
        dwLowDateTime;
    long t = l | h;
    return t;
}
```

This is an exercise in bit manipulation and proved to be tricky to get right, but if you type it in as shown it will work.

CONTEXT MESSAGE

With the parts of the IStoreFolder interface we need now in usable form, we can move on to design the YHM application. There are many ways of implementing this function, but a System Tray utility seems ideal because it can sit unobserved until it needs to attract attention. It is easy to create System Tray applications. Start a new C# project called YHM and add a form called Main. Drag and drop a NotifyIcon and a ContextMenu control from the toolbox on to the new form. To make the two controls work together, you need to set the NotifyIcon's ContextMenuStrip property to the name of the ContextMenu you have just placed on the form. You also have to assign an Icon to its Icon property because otherwise nothing shows in the System Tray when you run it. If you are using C# 2005 Express, the bad news is it doesn't come with an icon editor. However, there are a few available for download as freeware or shareware. The icon used here is included as part of the project on the cover disc.

Setting up the right-click context menu couldn't be easier. If you select the ContextMenu control, a menu appears that you can edit directly. For our program, we'll type in just one menu item: Exit. You can add others as you develop the project. Double-clicking on any of the menu items automatically generates a click event handler for it.

We also need this new form to be the startup form for the project. To do this, open the file Program.cs using the Solution Explorer and change the main method to read:

```
static void Main()
{
    Application.EnableVisualStyles();
    Application.Run(new Main());
}
```

The Main form contains the code that checks the email folders, but it is not intended to appear onscreen itself. If you want to hide it, change its Constructor to:

```
public Main()
{
    InitializeComponent();
    TopLevel = false;
}
```

Setting a form's TopLevel property to false means it is owned by a parent form. However, if the form's Parent property is null the form just vanishes and its Paint method is never called by the system, hence it becomes invisible.

We use the Main form's Load event handler to display the icon in the System Tray:

```
private void Main_Load(object sender,
    EventArgs e)
{
    notifyIcon1.Visible = true;
}
```

The only context menu event handler we need to implement is for the Exit option. This needs to remove the application from the System Tray (by making the icon invisible) and terminate the application:

```
private void exitToolStripMenuItem_
    Click(
        object sender, EventArgs e)
{
    notifyIcon1.Visible = false;
    Application.Exit();
}
```

If you don't make the icon invisible, it hangs around even after the application has finished.



Now we can start to build the code that inspects the mailstore and signals when a mail comes in. For this demonstration, a form is used to display an URGENT EMAIL message when the waited-for email arrives. You can change the code to respond to any email that you like.

We'll use the default Form1 to display our pop-up message. Add a multi-line text box to Form1 and change its Design, Modifiers property from Private to Public. This makes it possible to access the text box from our hidden Main form that contains the workings of our program.

Back in the Main form's Load event handler, we create an instance of Form1 and make it invisible:

```
f=new Form1();  
f.Visible=false;
```

This is also the place to create the OE object we need to work with the message store. To do this, we need to include the OE.cs file – which contains all the interface definitions and classes – in the project, and add to the start of the Main.cs file a reference to the relevant namespace:

```
using OE1;
```

Now we can create an OE object:

```
oe = new OE();
```

We also need to record the time that YHM was started, because we want to react only to emails that arrive after the start time:

```
StartTime = DateTime.Now;
```

Finally, we need a mechanism to check the email store for new emails at regular intervals. This is most easily achieved by adding a Timer control to the form and setting its Interval property to the number of milliseconds we want to leave between checking for new mail. For a check every minute, add:

```
timer1.Interval = 1000*60*1;  
timer1.Enabled = true;  
}
```

This code concludes the form Load event handler, but it is important to remember to add some variables we have used at the top of the

Main form class definition so that they have global scope:

```
private OE oe;  
private DateTime StartTime;  
private Form1 f;
```

THE TIMER

All the real work is done in the Timer's tick event handler. It's surprisingly simple, because most of the work has been done in preparing the foundations of the entire project. First we disable the timer so it doesn't trigger again until we have finished:

```
private void timer1_Tick(  
    object sender, EventArgs e)  
{  
    timer1.Enabled = false;
```

Next we use the OpenSpecialFolder method to open the inbox:

```
OEFolder Folder =  
oe.OpenSpecialFolder(  
    SPECIALFOLDER.FOLDER_INBOX);
```

The constant FOLDER_INBOX is defined in the OE.cs file and you can use other constants to open any of the special folders. As we discussed above, the method returns an object of type OEFolder, which represents the Inbox. To find out if there are any emails, we use the Folder's GetFolderProps method. This returns a FOLDERPROPS structure exactly like that from OE's GetNextFolder method:

```
FOLDERPROPS fp = Folder.  
    GetFolderProps();
```

We'll start by displaying the number of emails in the folder on our pop-up form:

```
f.textBox1.Text=f.textBox1.Text+  
    fp.cMessage.ToString();
```

We can only refer to textBox1 in this way because its Modifiers property was changed from private to public, which is a quick and slightly dirty way of displaying data. If there are any emails, we need to find out what time they arrived and who they are from:

```
if (fp.cMessage > 0)  
    {UInt32 ehandle = 0;
```

```
MESSAGEPROPS mp;  
do  
{mp= Folder.GetNextMessage  
    (ref ehandle);
```

The GetNextMessage method simply returns a MESSAGEPROPS structure until there are no more messages. When this happens, the entire structure is zeroed, so below we terminate the do-while loop when mp.MessageId is zero.

We check the DisplayFrom and Received fields of the MESSAGEPROPS structure to see if each message is the one we are waiting for:

```
if(mp.DisplayFrom=="mike james"  
    &&  
    mp.Received > StartTime)  
{f.Visible = true;  
    f.textBox1.Text = "You Have  
    URGENT Email";  
    f.Refresh();  
}  
}
```

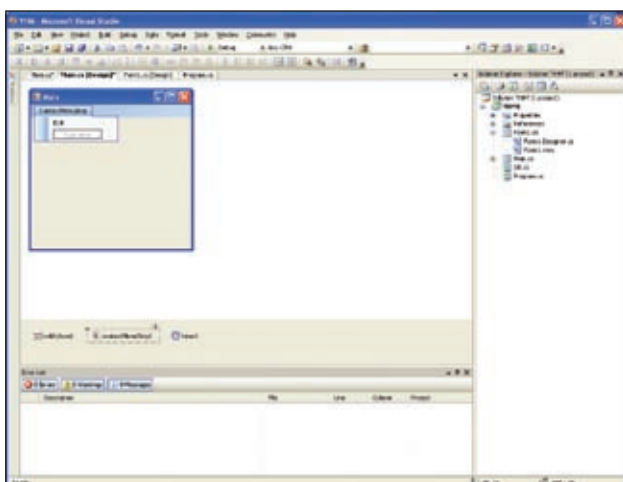
We've hard-coded a single name to look for. It wouldn't be hard to customise the project to check against a list of names or against a name entered by the user. The DisplayFrom field is the name as displayed in Outlook Express – that is a nickname, and not the full email address. If a new email is found from the designated user, the form is made visible, used to display a message and refreshed to make sure its display is up to date. If you enjoy setting yourself a challenge, you could insert code here to alert the user in a clever way, such as via SMS or an automatic phone call.

Finally, we will finish the loop and restart the timer:

```
}  
while(mp.MessageId>0 )  
}  
timer1.Enabled = true;
```

That's all there is to it. However, it's important to realise that all you are doing is inspecting the Outlook Express message store. You can do this even when Outlook Express isn't running but if you do, don't be surprised not to see any new emails. In practice, you'd want to add some code to start Outlook Express running before you start looking for new emails. In addition, there's no point checking for changes in the mail store every minute if Outlook Express is set to pick up mail only every 10 minutes. You need to match the rate that mail is collected to how often you check for changes in the message store. Finally, remember that no error handling is included in this example, and errors do occur if the mail store is in use by Outlook Express. You need to add some code that tests for problems and tries again in a few minutes.

All these enhancements are straightforward. To start Outlook Express, you can either use the Interaction.Shell method or implement more of the API, IO OutlookExpress. You can discover or set the time interval that mail is picked up via the Registry; the relevant key is Poll For Mail. Once you see how easy it is to work with the mail store and inspect the emails it contains, you are sure to think of other uses. However, if you want to go further, you'll need to complete more of the API – with both interface definitions and classes to wrap them. **CS**



◀ You simply type menu items on to the context menu